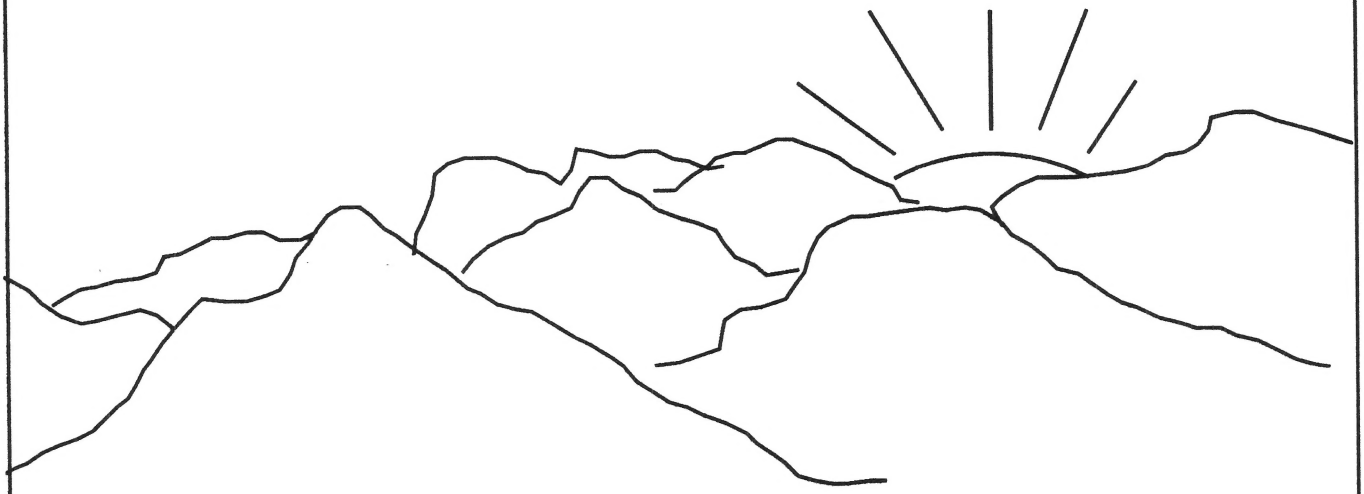
Implementation of Trellis

by:

Charles Doucette
George Fredericks
and
Mani Subramanyam

Colorado Center for Advanced Studies
Database Systems



digital Digital Equipment Corporation

Version 1.0
January 31, 1989

Acknowledgements

We would like to thank everyone in the Trellis Group for their help in writing this document. The Trellis Object-based System and Programming Environment is entirely the work of the Object-based Systems Group, which is now a part of Database Systems. This document describes the system internals. Special thanks to Mike Kilian, Bruce Bullis and Craig Schaffert for spending time with us explaining the current system. Thanks to Dave Wecker for getting us started on DECwrite, and to various people within the Colorado Center for Advanced Studies for their encouragement and support.

Table of Contents

1 Introduction 7

1.1 Overview of this document 8

2 Storage Representation of Objects 10

2.1 Storage Units and Address Spaces 10

2.2 A Trellis Object 10

2.3 A Trellis Large Object 11

2.4 Method, Code Vector and Environment Vector Objects 12

2.5 Other Objects 12

3 Storage Allocation 13

3.1 VAX Virtual Memory 13

3.2 Trellis in VAX VM 14

3.3 Trellis Workspace 14

3.4 Trellis Heap 14

3.5 Object Allocation 15

3.6 Reusing Objects with a Free List 17

4 Garbage-Collection 18

4.1 Overview of Garbage Collection 18

4.1.1 Heap Fragmentation and Reclamation of Free Space 18

4.1.2 Garbage Collection Phases 19

4.1.3 Garbage Collection Table 20

4.2 Reorganize and Combine Segments 20

4.2.1 Reorganize All Segments 20

4.2.2 Combine Adjacent Segments 21

4.3 Mark Phase 21

4.3.1 Introduction 21

4.3.2 Root Objects 21

4.3.3 Overview 22

4.3.4 Special Processing 23

4.4 Redirection Phase 23

4.5 Between Phases 23
4.6 Relocation Phase 24
4.6.1 Relocation Table 24
4.6.2 Overview 24
4.6.3 Pointer Redirection 25
4.7 Post Processing 25
5 Compilation 27
5.1 The String Canonicalization Table 27
5.2 The Compilation Process 29
5.3 Compilation States and Actions 30
5.4 The Type Checking Action 31
6 Execution 33
6.1 Operation Implementation 33
6.2 Register Usage 34
6.3 Argument Passing 35
6.4 Return Values 35
6.5 Preserving State 36
6.6 Operation Invocation 36
6.7 Operation Apply 38
7 Activities and Interrupts 40
8 Builtins 41
8.1 Builtin Operations 41
8.2 Builtin Fixed Names 42
8.3 Intrinsic Functions 42
8.4 World Variables 43
9 Callout 44
9.1 Callout Buffer 44
9.2 Callout Facility 45
9.3 Call-In 46

10 Documentation 47

11 References 49

List of Figures

1-1 The Trellis System 8

2-1 A Trellis Object 11

2-2 A Trellis Large Object 11

3-1 Organization of VAX Virtual Memory 13

3-2 Organization of Trellis in VAX VM 14

3-3 Segments of the Trellis Heap and their Areas 15

3-4 Linked List of Segment Descriptors from an Area Descriptor 15

3-5 Definition of a create(mytype) operation 16

3-6 Invocation of a create(mytype) operation 16

4-1 Two adjacent segments and their descriptors before combining 20

4-2 Two adjacent segments and their descriptors after combining 21

4-3 Weak Objects and Their References Before and After Marking 22

4-4 Organization of the Relocation Table and "Bad" Objects 24

5-1 The String Canonicalization Table 27

5-2 Example Type Module 29

5-3 System Data Structures for a Definition 29

5-4 Definition States and Actions 30

6-1 Three Kinds of Objects to Implement Operations 33

6-2 Register Usage 34

6-3 Code to Pop n Arguments off of the Stack 35

6-4 An Operation Invocation in Trellis 35

6-5 Partial Listing of Generated Code for an Operation Invocation 35

6-6 Code Generated for a Direct Call 36

6-7 Standard Prolog for a Method Object 36

6-9 Stack States 37

6-8 Code to Allocate Local Variables 37

6-10 Code to Return From an Operation 38

8-1 An Ordinary Operation Definition 41

8-2 Specification of a Builtin Operation 42

9-1 C code to invoke a Trellis operation (via call-in) 46

1 Introduction

This document discusses the implementation of the Trellis[1] runtime system. This is intended for use by developers and software engineers involved in building an object based database management system that makes use of the Trellis language and runtime system. The document is also intended to complement other technical reports that are currently available on Trellis (please see the bibliography at the end). The discussion here will be at the level of various components and subsystems of the Trellis runtime system, the algorithms and data structures used in their implementation and various design decisions that were made. Hopefully, this well help new developers and engineers to learn the "guts" of the present system very quickly.

The Trellis runtime system supports a programming environment that may differ from what most programmers are used to. In the usual programming environment, a programmer creates source files using a text editor. Then they are compiled into object files by a compiler. The object files may subsequently be placed into an object library. Finally, an executable image is created by a linkage editor. The tools used for these functions, editors, compilers, librarians, and linkers are generally invoked through the operating system's command interpreter. The resulting executable image can then also be invoked through the operating system's command interpreter.

The Trellis runtime system and programming environment provide a different paradigm for the the creation and invocation of programs. Source modules may still be created in a text editor. But the other functions needed to compile, link and execute programs are provided through the runtime system. The runtime system consists of a single program image (eg. OWLSYS.EXE, that runs within a VAX/VMS process). Trellis user programs are not separate images but instead are part of a workspace file that is mapped into the virtual memory of the Trellis runtime system process.

New code modules are added to the in-memory workspace by the Trellis compiler which is part of the runtime system. Source programs can be created and debugged with a set of interactive programming environment tools that are themselves Trellis application programs running within the runtime system. The Trellis compiler can be invoked interactively from the programming environment or can be used directly when the runtime system is invoked from the operating system's command interpreter. The in-memory workspace can then be written back to a new workspace file.

A Trellis user program can be initiated interactively from the programming environment by invoking an *operation* or *method* that exists in the workspace. These are similar to functions or routines in conventional programming languages. Alternately, the user can supply a "main" method at compile time that will be given control after the runtime system loads a workspace. Figure 1-1 shows a high level decomposition of the subsystems in the Trellis system.

[1]Trellis is a trademark of the Digital Equipment Corporation

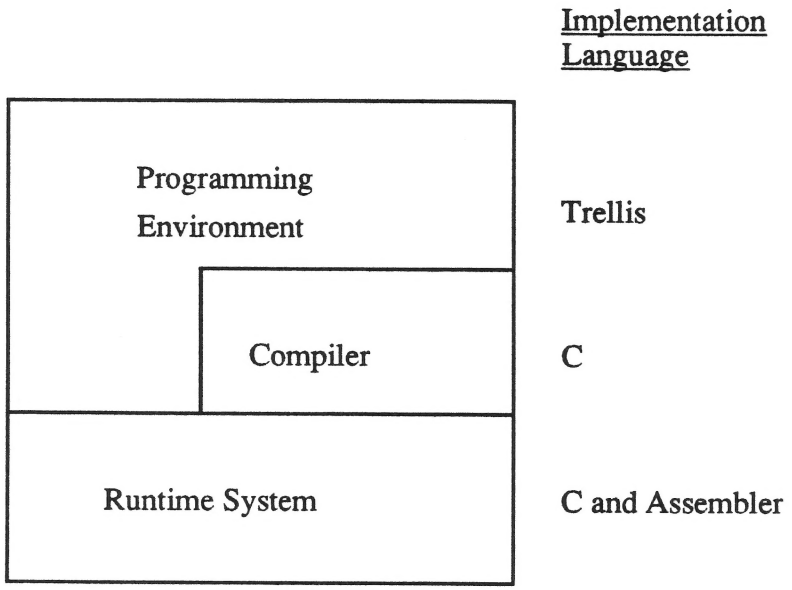


Figure 1-1 The Trellis System

The current implementation of Trellis runs on the VAX on VMS and Ultrix. The design and implementation is intended to make this a portable system, that can be retargeted to different hardware platforms.

1.1 Overview of this document

We now describe briefly what each of the following chapters cover. Chapter 2, *Storage Representation of Objects,* introduces terminology to be used for components of the underlying architecture. It covers the representation of regular and large objects, representation of integers, and the motivation for the particular scheme chosen for their representation, method objects, code vector objects and environment vector objects. It also introduces certain other kinds of objects.

Chapter 3, *Storage Allocation*, covers the Trellis system organization and storage allocation techniques and structures that are used by the Trellis system. It first introduces VAX virtual memory concepts. It then describes the Trellis system organization within VAX virtual memory, the Trellis Heap and its components : areas, segments, free lists, area descriptors, the area index table and chunks(which are storage units of 64K bytes each). It also deals with the layout of the programming environment, the runtime-system and builtins. It identifies the components that are part of the workspace and the ones that are a part of the run-time system. It gives an example of allocation of a new object, the create calls that are necessary and presents pseudo code for the space allocation procedure (involving segments etc.).

Chapter 4, *Garbage Collection*, covers the garbage collection process in Trellis. It looks at the two options available to the user, (i.e. garbage-collect area T or garbage-collect all areas), and shows the sequence of operations involved: reorganization of segments, consolidation of adjacent segments in the same area, actual garbage collection with the mark, redirect and relocate phases. It describes the multi-purpose gc-table, and covers post-processing.

Chapter 5, *Compilation*, deals with compilation issues. It covers the Trellis incremental compiler, symbol tables that are used by the compiler, parsing and code generation. It also deals with how strings are stored and manipulated, the string canonicalization table, source code objects and how they are set up and used. It looks at how the symbol tables are organized to keep information about the type hierarchy, inheritance and secret inheritance, and precedence rules for operations present in multiple types.

Chapter 6, *Execution*, deals with code execution. It presents code fragments that are generated for methods, shows the flow of control between an operation's method object, code vector object and environment vector object. It also tracks the changes in the stack frame It presents implementation details of "apply" and "cached apply"(these are used to speed up a method invocation).

The Trellis Activity System is the subject of chapter 7, *Activities*. Activities are multiple threads of execution within the context of the Trellis process. We first introduce the concepts, give an example of how they are available in the Trellis language. We then look at the implementation in terms of the activity scheduler, activity stacks, the "activity" object, multiple levels or priorities, wait-queues, locks and interrupts. We also explain the semantics at the level of trellis code versus that at the level of the run-time system.

Chapter 8, *Builtin Operations*, deals with builtin types in Trellis. It tries to answer questions such as: *What are they? Where do they reside? When is it appropriate to use them? What are the categories of builtin operations?* We also cover here the builtin jump table, show a code fragment of a method that calls a builtin, and trace the flow of control to the builtin.

Chapter 9, *Callouts,* covers the callout facilities. It describes how callout is implemented. It looks at why callouts are necessary.

2 Storage Representation of Objects

The unit of encapsulation in the Trellis System is the "object". This chapter deals with how objects are represented in memory. We will first try to define a standard set of terms for the units of storage. We will also specify certain assumptions made by the Trellis system about the underlying hardware architecture (which have affected the Trellis design). While many of these assumptions are specific to the VAX, Trellis is intended to be easily portable to other hardware architectures. Hence we will keep our discussion general wherever possible.

2.1 Storage Units and Address Spaces

The Trellis System runs on a 32-bit hardware architecture. A *word* is two bytes(where a byte is the usual 8-bit entity). A *longword* is four bytes (or 32 bits in size). A longword (32 bits) is required to specify the address of any byte of virtual memory. The high bit (or bit 31) of the 32-bit virtual address is used to distinguish between a process virtual address and a system virtual address. A user process (such as Trellis) can operate only in the process virtual address space (with bit 31 clear). Further, bit 30 divides the process virtual address space into the program region, called P0, indicated when bit 30 is clear, and the control region, called P1, indicated when bit 30 is set.

2.2 A Trellis Object

A Trellis Object is longword-aligned, and may have references to other objects, as well as raw bytes.The references are 32-bit longwords, and are direct virtual addresses of other objects. Thus an object A that has a reference to another object B has B's direct virtual address (a 32-bit longword). A pointer that has its high bit set is interpreted as an immediate 31-bit signed integer value. This was a storage optimization done so that integer values do not need a separate object to represent them, and they can be accessed faster. The tradeoff here is that integer operations in Trellis need to account for the range shift, and hence typically involve more machine instructions than conventional languages generate for a similar operation.

The first element in a Trellis object is a reference to a class object that contains a type description or a 'template' of this object itself. The class or type reference is followed by two fields that specify the count of raw bytes that follow, as well as the count of object references that are present in this object after the array of bytes. The object is finally padded with 0-3 bytes to make the *next* object longword aligned. The Trellis runtime system has a number of different object representations. The most common one is shown in Figure 2-1. Objects with the format given in Figure 2-1 have references or descriptors that are 32 bit virtual addresses. Since the count fields for bytes and references are 16 bit signed integers, an object represented in this format could contain as many as 32K bytes of data plus 32K references.

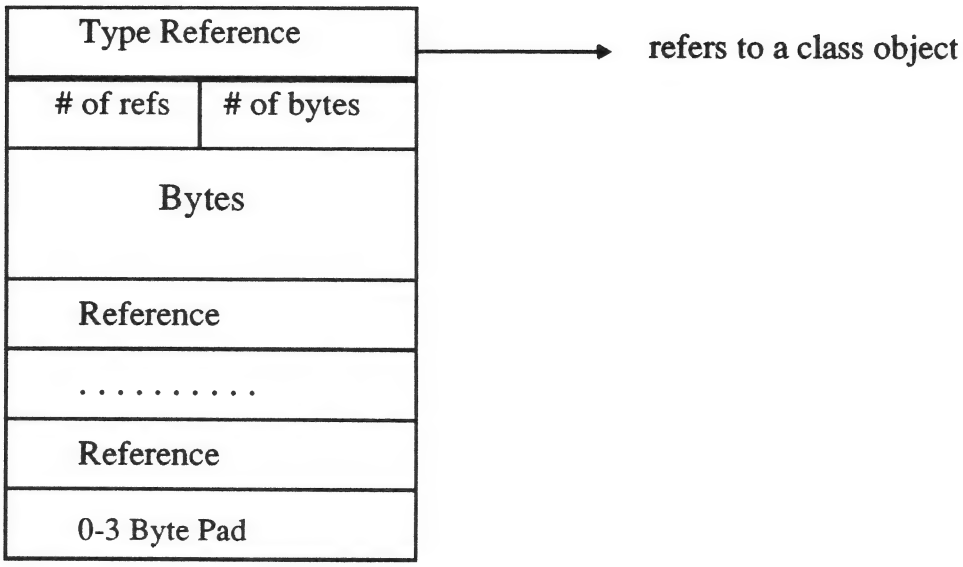


Figure 2-1 A Trellis Object

2.3 A Trellis Large Object

Figure 2-2 is a representation for *large objects* in Trellis. This is used when the regular object is not large enough.(i.e. has more than 32K bytes or references).

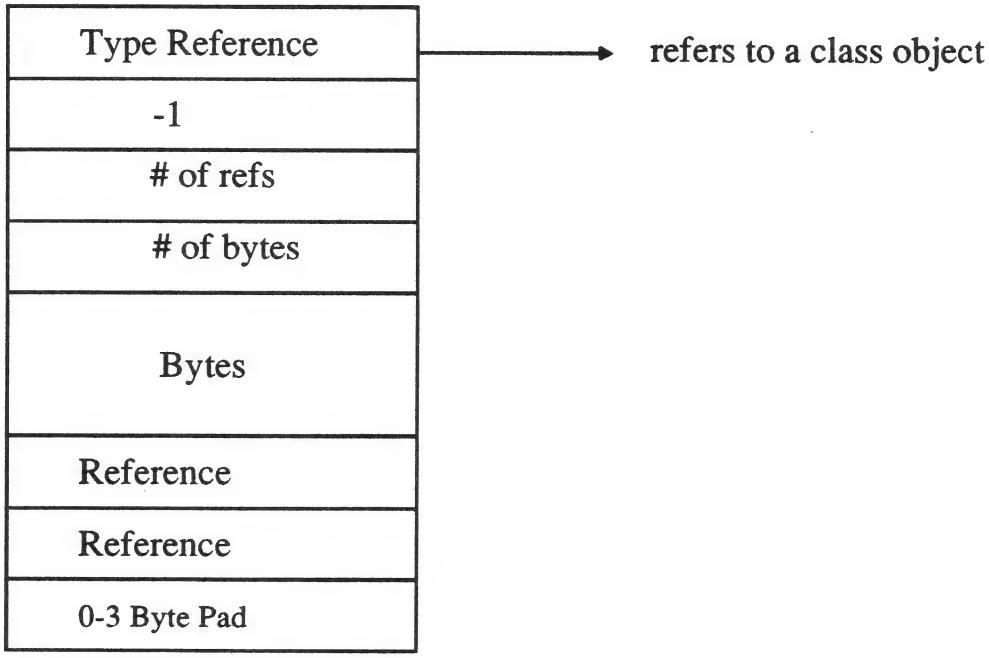


Figure 2-2 A Trellis Large Object

It provides 32 bit integers to describe the number of bytes of immediate data and the number of references to other objects.This representation can be distinguished from the first object representation by the fact that the second integer after the type reference has the sign bit set (is less than zero). Note that the existence of this additional object format does not mean that every object requires a format test before use. Only certain code that works for both kinds of objects must do this test. Object specific code will be generated without the need to contain such a test because the compiler will know the object format used.

2.4 Method, Code Vector and Environment Vector Objects

We now describe some other Trellis objects that will be referred to in later chapters quite often. These are Trellis *method* objects, *code vector* objects and *environment vector* objects. Programming in Trellis involves writing type modules. Type modules consist primarily of components (or fields) and methods (or operations). Consider a Trellis operation:

```
operation foo (me, a: type_x, b: type_y)
   returns (type_z)
    is
    begin ...
       p (x, y);
      q (l, m); ...
    end;
```

Associated with the operation foo will be a method object that references a code vector object and an environment vector object. All *callers* of the operation foo will invoke it through a reference to foo's method object.The environment vector object is a vector of references to other method objects. The code vector object contains code that is required for each invocation of a method in the method currently being executed. (i.e. in method foo's code vector object we have code that invokes operations p and q. This code pushes the calling arguments on the stack, sets up the pointer to p's method object and makes the subroutine call.). Separating the bytes and the references for a method invocation in the code vector and environment vector objects helps in easy location of references for fixup during garbage collection. We will look at these objects and show the complete flow of control for a method invocation in the chapters on compilation and code execution.

2.5 Other Objects

The Trellis compiler generates and uses a number of objects, such as source code objects (which store the source code in a special way), symbol table objects etc. We will look at these in detail in the chapter on compilation. A string in Trellis is stored as a set of bytes (the characters), in a single object. Integers, as explained earlier, do not have a separate object of their own, but instead are stored in the references slots with the high bit set.

Finally, some general comments about Trellis objects. Most objects are fairly small. The typical object would be less than 100 bytes. Most of the contents of the typical object created by the user would consist of references to other objects.

3 Storage Allocation

As we have described, the Trellis system is made of many components. Besides describing how to dynamically allocate storage for these components, we will describe the initial allocation of storage for these components. However, first we must reintroduce VAX virtual memory to see how the storage is organized for any user process.

3.1 VAX Virtual Memory

VAX virtual memory is divided into two virtual address spaces, system space and process space. Each virtual address space contains two virtual segments (for a total of four) with 2^30 bytes each. These segments are numbered and named to designate the process or system virtual address space: P0, P1, S0, and S1.

A virtual memory address (pointer/reference) is represented in 32 bits (a VAX longword). The most significant bit (bit 31) of a virtual address is used to designate the address space (0x = process address space and 1x=system address space). The next most significant bit (bit 30) is used to designate the segment within that address space (x0=segment 0 and x1=segment 1). Therefore, the two most significant bits (bits 30 and 31) in a virtual address designate the following segments: 00=P0, 01=P1, 10=S0, and 11=S1. The least significant 30 bits (bits 0 through 29) are used for addressing within a segment.

The operating system and the page tables (which are used to translate virtual memory addresses to physical memory addresses) are stored in segment S0. Segment S1 doesn't really exist (even as a virtual segment) - it is "reserved" for future use. S0 is never context-switched (it is always resident).

Each process has its own copy of the P0 and P1 segments because they are context-switched. P0 contains the code and all compiler allocated storage (initialized and uninitialized data) for an executable. The heap (where dynamic data is allocated) starts at the end of this area in P0 and grows towards the base of P1. The stack starts at the limit of P1 and grows downward towards the limit of P0 (the base of P1).

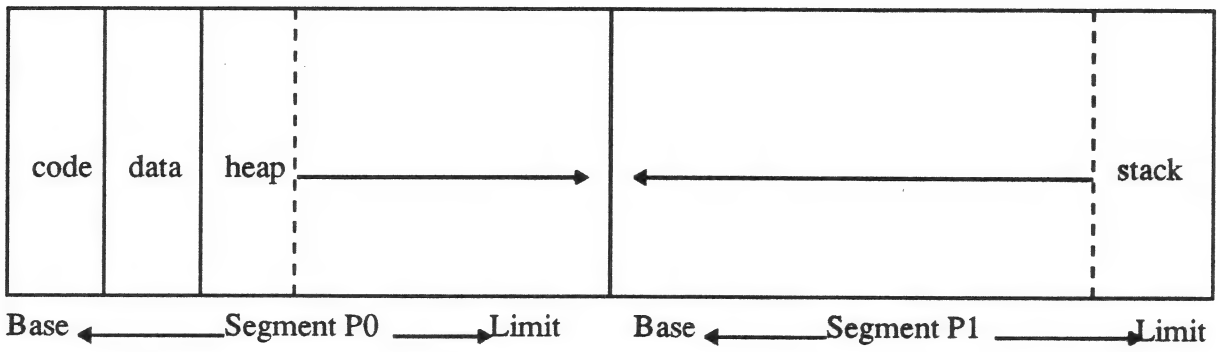


Figure 3-1 Organization of VAX Virtual Memory

3.2 Trellis in VAX VM

The Trellis runtime system, code for the builtin operations, and the activity stacks (see chapter 5, *Activities*) etc. are stored in the P0 segment. The Trellis heap is stored in the P1 segment below the system stack (it starts at a fixed location and grows downward - leaving a certain amount of room for system stack growth and the VAX/VMS debugger). Any of these components can and do dynamically allocate more memory. For instance, to call-out to a routine written in another language, that routine is dynamically linked to the Trellis runtime system's executable code (for more details see chapter 8, *Call-Out)*. Activity stacks can be allocated and deallocated during a Trellis session and the Trellis heap can also grow (it currently doesn't shrink).

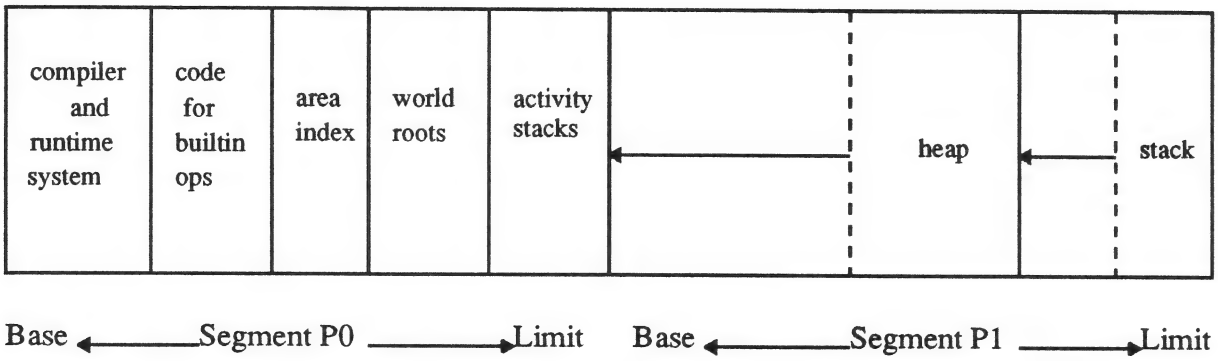


Figure 3-2 Organization of Trellis in VAX VM

3.3 Trellis Workspace

Trellis uses the "workspace" model for object persistence. A Trellis workspace contains some references to root objects (the "World_Roots"), some information about the Trellis Heap (the area index), and the Trellis heap itself. The Trellis heap (part of the Trellis workspace) contains all of a user's code and data objects as well as the Trellis programming environment (also made of code and data objects). At the beginning of each Trellis session, the Trellis workspace is mapped into a fixed location in virtual memory (in the P1 segment as described above). Pages (512 bytes) are copied from the workspace file to memory as necessary (read-only). If a page of memory (which is part of the workspace) is modified, it is written to a system page file (not back out to the same workspace file). After each Trellis session, all of the Trellis workspace is written to a new version of the workspace file. When a user begins a Trellis session, he can either resume with the latest version of the workspace file; or, he can restart with a previous version of the workspace file.

3.4 Trellis Heap

The Trellis heap is divided into areas. This was designed to improve the locality of object references and thereby minimize page faults. This is accomplished because there are different categories of objects. Some objects contain just bytes (e.g. code vectore objects), some contain just ob-

ject references (e.g. environment vector objects); and further, the objects referenced can usually be classified into similar categories (i.e. put objects which reference each other into the same category).

Consequently, the heap is divided into *four* different areas where these categories of objects are allocated: area T (temporary), area B (bridge), area R (runtime), and area C (compiler). The only restrictions on object references between areas is that area B is the only area that can contain references to objects in area T (this limits the scope of garbage collection and essentially is the justification for the existence of a bridge area).

The Trellis heap is divided into segments (allocated in 64K byte chunks) and each segment is part of a particular area. All segments in an area are not necessarily contiguous. When the Trellis heap (and workspace) is first created, it is just raw bytes of memory. When a segment is needed, these bytes are type-cast to a segment descriptor, appropriately initialized, and linked into the list of segment descriptors for the desired area.

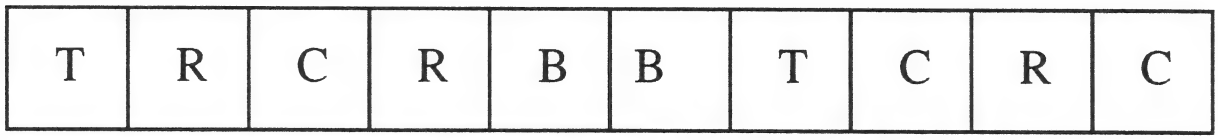


Figure 3-3 Segments of the Trellis Heap and their Areas

A segment descriptor contains three fields: a pointer to the next segment descriptor, a pointer to the first byte of free space in the segment, and a pointer to the last byte of a segment.

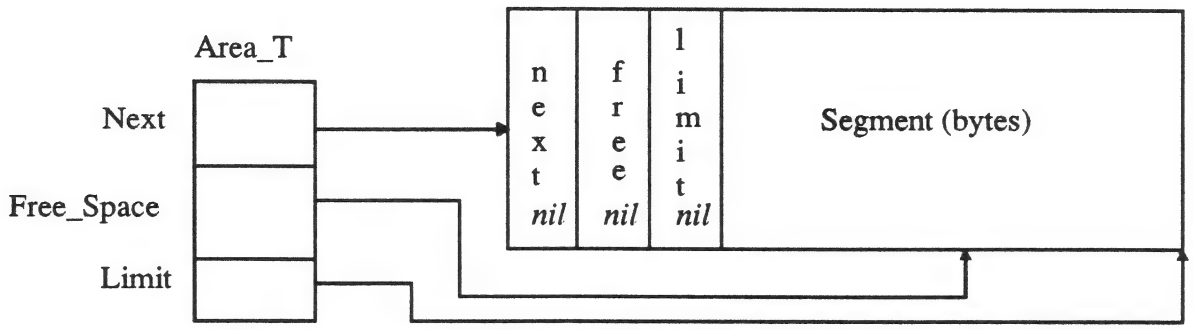


Figure 3-4 Linked List of Segment Descriptors from an Area Descriptor

Objects are allocated from the free space in the most recently allocated segment for a particular area. If there is not enough room in this particular segment, another segment is allocated in the Trellis heap (perhaps by using existing raw bytes already allocated to the heap). Then, the object is allocated from the free space in this newly allocated segment.

3.5 Object Allocation

For a Trellis user to create (allocate) an instance of an object of a particular type, he must write a create operation as part of a type module and then invoke this create operation. An example of each is shown in figures 3-5 and 3-6.

```
type_module Version

component me.name : string
        is field;

operation create(MyType, s:String )
        returns ( MyType )
        is allocate
        begin
                me.name := "\"" & s & "\"";
        end;

end type_module;
```

Figure 3-5 Definition of a create(mytype) operation

The Trellis compiler will turn this definition of a create operation into two operations: create(mytype) and create(me). The create(mytype) operation will call "Allocate()" and then call create(me). Allocate() will allocate an object in the temporary area with the right size and initialize the newly allocated object's header with the proper count of bytes, references, and the object's class pointer (a pointer to the type "Version"). The create(me) operation will initialize all of an object's components (in this case the component *name*).

```
var v : Version := create(Version,"1.0");
```

Figure 3-6 Invocation of a create(mytype) operation

To be more precise, each create(mytype) operation calls a core allocate function. These functions take a pointer to the class object and either just a count of object references (as a regular integer) or a count of bytes and a count of object references (represented as Trellis integers) and returns a pointer to a newly allocated object in the temporary area. Because the arguments and result values are passed in registers and the necessary calculations are performed in a few instructions, object allocation usually completes quickly. If the object can not be allocated in the most recently allocated segment (which is part of the temporary area), the actual function Allocate() is called.

This function Allocate() takes a reference to the class object, the count of bytes, and the count of object references and returns a pointer to an allocated object. It selects the proper (the most recently allocated) segment which is part of the temporary area, and tries to increment that segment's free space pointer by the total number of bytes needed to allocate the object. If the result of this calculation is a pointer outside of the segment (past the segment's limit), World_Expand is called to allocate a new segment for this area (the temporary area). Once a segment is found with enough free space to allocate this object, the free space pointer is incremented accordingly (decreasing the amount of remaining free space) and the new object's header is initialized (with the class pointer, and the number of bytes and object references) and the large object flag is set if necessary.

3.6 Reusing Objects with a Free List

When the compiler doesn't need an object anymore, it places the object on a free list (one is maintained per area) to avoid accumulating garbage objects and therefore avoid invoking the garbage collector. When the compiler needs a new object, it will try to find a discarded object on the appropriate area's free list. If there are no objects on this free list, the appropriate object allocation function must be called.

Realize that a Trellis user allocates instances of objects explicitly (by invoking a create operation); but, deallocation is managed implicitly by the garbage collector.

4 Garbage-Collection

The term garbage-collection is used to describe the process of reclaiming storage. If an object is not referenced by any other object in the Trellis heap (i.e. it is garbage), the storage that it is using can be reclaimed (i.e. collected) and later allocated to a new object. If storage is never reclaimed, a user might run out of virtual memory in the Trellis heap and not be able to allocate new objects. Garbage-Collection is an alternative to explicitly de-allocating certain objects. It is a safer process because it ensures that there are no references to an object before its storage is reclaimed.

The Trellis heap is made of segments (allocated in 64K byte chunks) which are part of areas. Objects are stored in segments. Currently, no objects or segments are de-allocated from the Trellis heap. In each segment, "good" objects are compacted (relocated to lower virtual addresses *in the same segment*) and *most* "bad" objects are either overwritten or left in (the segment's) free space (as opposed to being placed on a free object list). The Trellis compiler actually manages its own storage by placing unused objects on an area free (object) list and trying to allocate objects from that area's free (object) list.

There is a "Storage Watchdog" tool which checks for available virtual memory in the Trellis heap at specified intervals. When that gets too low, a window is popped up which suggests that you either garbage-collect the temporary area or all areas. Garbage-collection doesn't happen automatically; it has to be initiated by the user. Below we shall review what actually happens when the user initiates either form of the garbage-collection process.

4.1 Overview of Garbage Collection

4.1.1 Heap Fragmentation and Reclamation of Free Space

In order to allocate an object in a particular area, you find the most recently allocated segment in that area and try to put the object in that segment's free space. If there is not enough room to place the object in that segment's free space, another segment is allocated for that area and the object is placed in the newly allocated segment's free space (which is initially the entire segment). This object (and segment) allocation process may leave many bytes of unusable free space in an area's segments (other than the most recently allocated segment).

Before the garbage collection process begins, the garbage-collector will try to take the necessary steps to reorganize each area's free space (i.e. undo the fragmentation). There are two actions the garbage-collector can perform to accomplish this. First, segments can be reorganized in the heap (to make all of an area's segments adjacent). Second, any adjacent segments in an area can be combined into one segment (which will also combine each segment's free space).

If the user chooses to garbage collect all areas, the garbage-collector will reorganize all the segments in each area (as necessary) to make sure that all those segments are adjacent. If the user only garbage collects the temporary area, this action is not performed. This allows reclamation of *all* the wasted free space in each area because *all* segments of a particular area will be consolidated into *one* (large) segment.

In order to consolidate the wasted free space in an area's segments, *adjacent* segments are combined (before the garbage collection process begins). Note that not all segments in an area are adjacent. Also, this happens whether the user chooses to garbage collect the temporary area or all areas. When two adjacent segments are combined, the free space of the first segment is turned into an object which is garbage collected (and eventually will be combined with the free space of the second segment - after compaction). Unfortunately, unless the newly consolidated segment is the most recently allocated segment for the particular area (which would be true if all the segments in the heap were reorganized and then combined), the combined free space is still wasted.

4.1.2 Garbage Collection Phases

The garbage-collection algorithm used in Trellis is a mark and sweep with compaction process. Below is an overview of the traditional *four* phases of this process:

1. *Mark* starts at known root objects and marks all accessible ("good") objects

2. *Locate* scans the Trellis heap, locates garbage (non-accessible or "bad" objects), and calculates new object locations (for the "good" objects).

3. *Fix-Up* scans all "good" objects and fixes-up object references to point at the new object locations (of the "good" objects).

4. *Compact* relocates all "good" objects thereby reclaiming the storage used by the "bad" objects).

The above scheme involves *four* linear scans of the Trellis heap (one per phase). To improve on the above scheme, Trellis uses only *three* phases for garbage-collection:

1. *Mark* starts at known root objects and marks all accessible ("good") objects

2. *Redirection* scans the Trellis heap, finds the garbage ("bad" objects), calculates new locations (for the "good" objects), and fixes-up *backward* references (references to "good" objects stored at a lower virtual memory address than the current object).

3. *Relocation* scans the Trellis heap, fixes-up *forward* references (references to "good" objects stored at a higher virtual memory address than the current object), and relocates the "good" objects.

Because this implementation of garbage-collection involves only *three* memory scans (one for each phase) instead of four, it completes execution much quicker than the traditional implementation described above. This reduction in phases is possible due to the use of a garbage collection table (described below).

Each scan of the Trellis heap will examine one applicable segment at a time in a linear fashion (from the lowest to the highest virtual memory address). A segment is applicable if it is in the set of areas to scan from or to collect in - depending on the phase.

4.1.3 Garbage Collection Table

A multi-purpose table is used in garbage-collection (used in different ways by different phases). For the mark phase, it is used as a stack for objects to be marked. This stack is conceptually a set (no duplicates). During other phases, it is used as a relocation table - given an object reference, find the object's new location.

4.2 Reorganize and Combine Segments

4.2.1 Reorganize All Segments

Before this operation begins, the garbage collector will try to "re-map" each segment. To accomplish this, it loops through each area and each segment in that area and calculates the proper location for the segment and puts that information into the garbage collection table. If the segment's actual location does not match the calculated proper location (obtained from the garbage collection table), then the segment is out of place and the reorganization of segments will happen.

To reorganize the segments, the garbage collector will first modify *all* object references (the world roots, the activity stacks, and the Trellis heap) to reflect any segments to be relocated. Then, it will do the relocation of all the segments which are out of place by using a temporary segment for swapping segments.

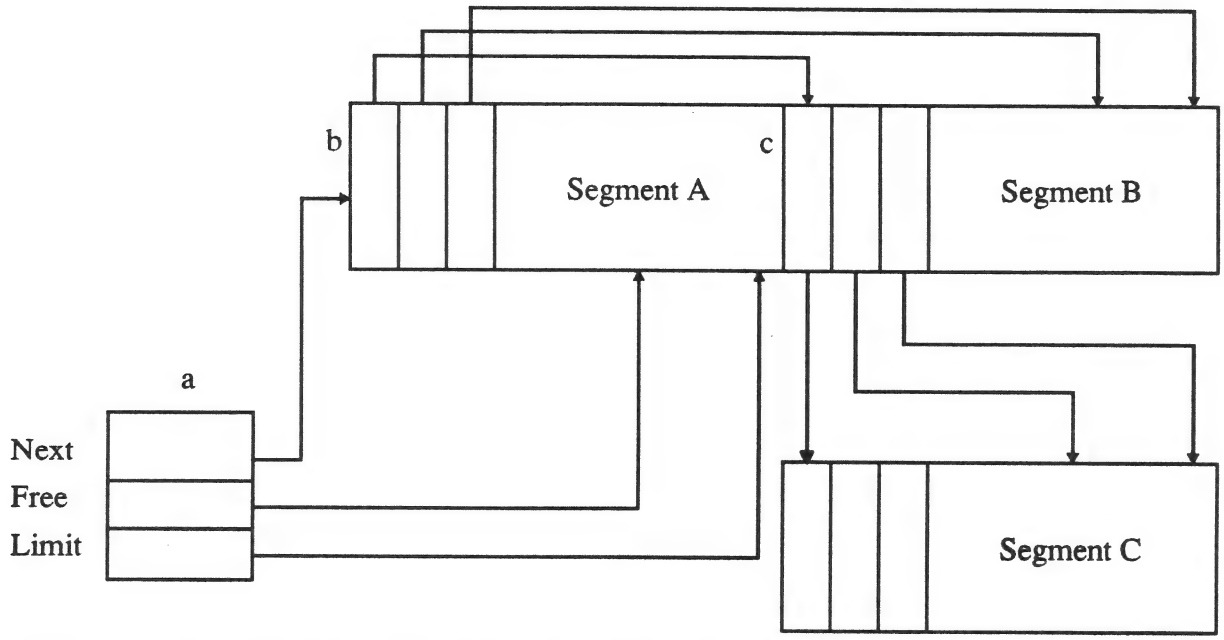


Figure 4-1 Two adjacent segments and their descriptors *before* combining

4.2.2 Combine Adjacent Segments

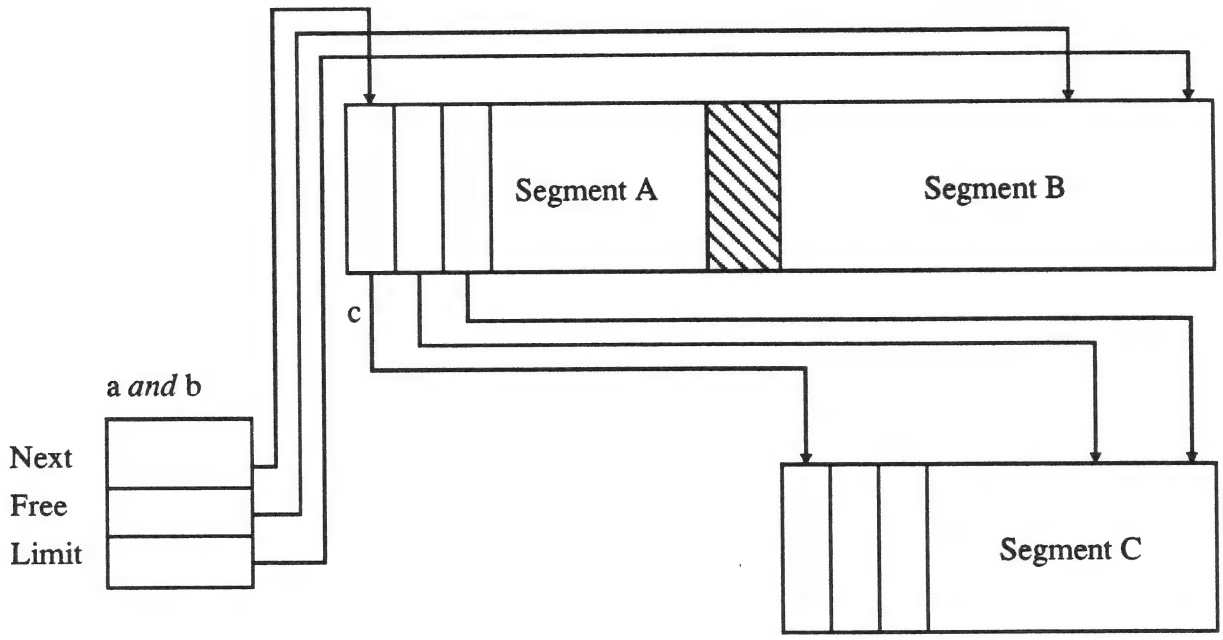


Figure 4-2 Two adjacent segments and their descriptors *after* combining

In figures 4-1 and 4-2 we see a few segments (A, B, and C) and their descriptors (a, b, and c) before and after combining. Basically, the segment descriptor for A (a) was changed to point to the free space in segment B and the limit for segment B. The segment descriptor for B (b) was changed to become the segment descriptor for C (c). Finally, the free space in segment A became an object (the rectangle with the pattern of diagonal lines) which will be garbage collected and after compaction will be part of the free space in segment B.

4.3 Mark Phase

4.3.1 Introduction

This is the phase where "good" objects are marked. An object is "good" if it is accessible (at least one other object contains a reference to it).

Because objects are aligned on longword boundaries, bits 0 and 1 of any object reference is free for manipulation by the garbage-collector. An object is marked by setting bit 0 (LSB) of the class pointer.

4.3.2 Root Objects

In order to mark all objects in the set of areas being collected, the garbage-collecter must start with references to some "root" objects. The mark phase starts from references to these sets of "root" objects in this order: "weak" objects, "world root" objects, objects in the set of areas to

scan, and objects referenced from activity stacks. "World Root" objects are those system types or classes such as: Declared_Type_Class, Method_Class, Fixed_Name_Class, and Component_Class etc.

"Weak" objects are those objects (like string canonicalization tables) whose *references* are "weak". Before the mark phase begins, all "weak" objects are marked; but, those objects referenced *from* the "weak" object are *not* marked. If at the end of the mark phase, objects that are referenced by a "weak" object aren't marked, those references are cleared (and since those objects aren't marked, they will be garbage-collected).

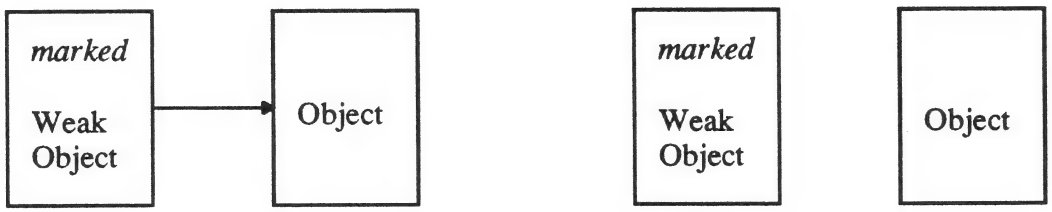


Figure 4-3 Weak Objects and Their References *Before* and *After* Marking

4.3.3 Overview

The top-level algorithm of the mark phase will iterate through all areas of the Trellis heap (in the set of areas to scan from), all the segments in that area, and all the objects in those segments. The first object in the first segment in an area to scan from is designated the current object. The garbage-collecter will use the information in the current object's header to determine the size of the current object which is how many bytes it must skip to get to the next object.

An attempt is made to mark each object which is referenced from the current object (including the object's class; and, the code and environment objects if the current object is a method). Objects which are referenced are not marked if they are outside the set of areas being collected or if they have already been marked. Note that the current object is not marked.

After an object is marked, the object's class pointer is put onto the "to mark" stack, the object's references are put onto the "to mark" stack, and the reference to the object that was just marked is *removed* from the "to mark" stack. If the current object is a method object, its references to the corresponding code and environment objects are extracted (from the bytes section) and are put onto the stack. Instead of putting *all* of the object references on the stack, the garbage-collector will put the address and count of the references onto the stack.

This part of the mark phase will recurse (depth-first) until the stack is empty and then try to mark from the next object. It ignores all references to marked objects and references to objects outside of the set of areas that are being collected (quickly determined with the area index). Small integers (pseudo references) are also ignored.

This process will find all references to objects in the areas to be collected and mark all referenced objects in that area. If there is a tree of objects in the area to be collected, and there are no

references to the root object of that tree, then only the root object will be collected (all other objects in the tree will be marked).

4.3.4 Special Processing

After the mark phase, the garbage collector will be modifying all references to objects in the areas to be collected. Object references are assumed to point at the beginning of the object; and, calculations of how far an object must be relocated are relative to the beginning of an object. So, certain object references that point inside objects are modified during the mark phase to allow uniform treatment of object references during the other phases. After the garbage collection process is complete, these special object references are restored to their original values. These special pointers are references to the code vector object from within a method object and saved PCs (the Program Counter is a VAX register) in activity stacks.

The references to the bytes section of code vector object from within a method object are changed to point at the beginning of the code object by subtracting the size of an object header. Saved PCs are changed to be offsets (small integers - psuedo references) from the beginning of the code vector object by subtracting the address of the beginning of the code vector object..

Between the redirection and relocation phases, the references from method objects to code vector objects are restored. After the entire garbage-collection process, the saved PCs are restored (see section 4.7).

4.4 Redirection Phase

This is the phase in which the garbage-collector determines where to relocate "good" objects and fixes-up any *backward* pointers. Backward pointers are those pointers which point to objects stored at a lower virtual memory address (in the Trellis heap) than the current object. These pointers can be fixed-up because the relocation table has already been filled in up to that point (all objects before the current object have been assigned a new location). *Forward* pointers are fixed-up in the relocation phase (after this phase fills in the entire relocation table).

This phase only iterates over segments that are in the set of areas to be collected because these are the only objects eligible for relocation. References to these objects from segments in other areas will be modified between the redirection and relocation phases.

Before this phase begins, the garbage collection table is cleared. It was used by the mark phase as a "to mark" stack and will be used by subsequent phases as a relocation table.

4.5 Between Phases

Between the redirection and relocation phases, all references from objects *outside* the areas being collected to objects *inside* those areas are fixed-up with the information from the completely filled-in relocation table (because now the garbage-collector knows where all the "good" objects will be relocated). The garbage collector iterates through the same set of objects (except those in the areas to be collected) that were used in the mark phase (i.e. the "World_Root" objects, the activity stacks, etc.).

4.6 Relocation Phase

In this phase, the "good" objects are compacted (relocated) and all the *forward* pointers are fixed-up (because the backward pointers have already been fixed-up in the previous phase).

4.6.1 Relocation Table

Segments in the Trellis heap are further divided into blocks (1K bytes) for the purposes of garbage-collection. The relocation table contains one entry for each block in the Trellis heap. Each entry in the relocation table contains the new address of the first "good" object in the block, the count of bytes of garbage up to that point in the block (before the first "good" object), and the offset of the next "bad" object in that block. Each "bad" object also has a cumulative count of the number of bytes of garbage seen so far in that block and the offset of the next "bad" object. These two pieces of information are contained in each entry of the relocation table and in each "bad" object and are called an "Info block".

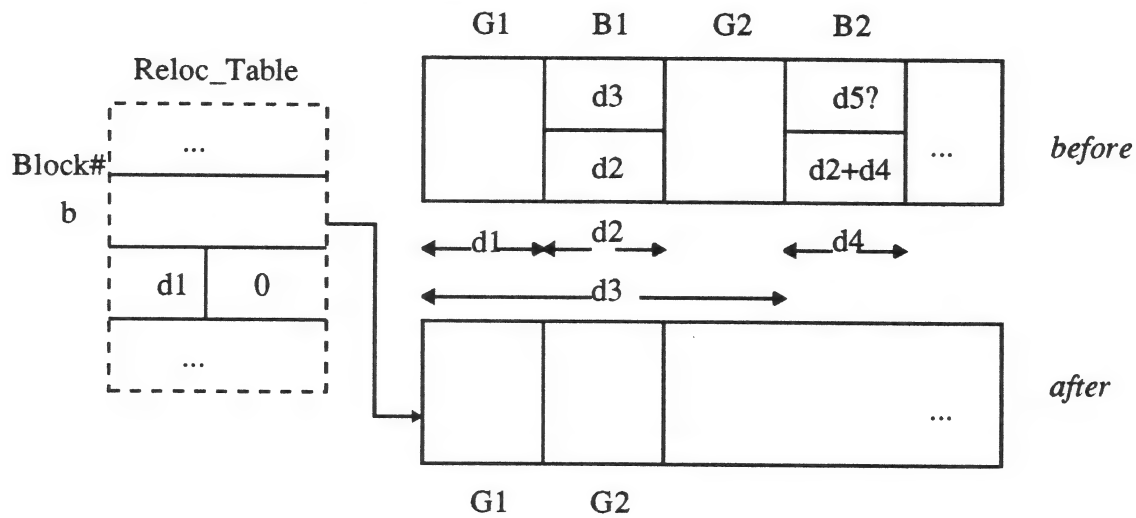


Figure 4-4 Organization of the Relocation Table and "Bad" Objects

4.6.2 Overview

The relocation phase iterates over each segment in an area to collect. Within each segment, the garbage-collector iterates over the blocks in that segment (the block number is used as an index into the relocation table). For each object in the block, a number of steps are followed which are described below. Note that references are only fixed if they are *forward* references.

1. If the object was marked, it is unmarked; else, go to the next object.

2. If the object is a method, fix the references to the method's class, the environment object, and the code object; else, fix the class pointer.

3. All other references from this object are fixed.

4. Finally, all contiguous "good" objects (from the first marked object to the next unmarked object) are copied to the new location (determined from the relocation table - see below).

4.6.3 Pointer Redirection

With Figure 4-4 in mind, let us determine how to redirect a reference p to an "good" (marked) object G2. Here is how the new address of p is determined from the relocation table (the pointer p is redirected):

1. Determine the block which G2 lies in - in this case, block number b. This is determined by shifting p right by 10 bits (BlockSize = 1024 bytes = 2^*10* bytes) leaving the upper 22 bits. This block number b is used as an index into the relocation table.

2. Determine the byte offset of G2 into block b by masking p with the BlockMask (BlockSize - 1 = 1023 bytes). In this case, the byte offset of G2 is d1+d2.

3. Determine the base address of block b by masking p with the inverse of the BlockMask.

4. Follow the chain of "Info blocks" (from the initial entry in the relocation table through the garbage-objects) until you find one whose Next_Block is greater than the offset determined above. So, in this case you find bad object B1 and it says the next bad object B2 is at offset d3 which is greater than the offset of object G2 (d1+d2). At this point the cumulative total of garbage-bytes (d2) will be correct for this object reference p (it will be a count of all the garbage in the block before object G2).

5. The new address for p is calculated by adding the new address of the first "good" object G1 (found in the relocation table slot for block number b) to the offset determined above (d1+d2) minus the amount of garbage seen in the block so far (d2). This leaves the new address of G1 plus the size of G1 (d1).

Here is some C code to show exactly what happens:

```
#define GC_BlockShift   10                  /* 2^10 = BlockSize*/
#define GC_BlockSize    (1 << GC_BlockShift)/* 1K = 1024 bytes */
#define GC_BlockMask    (GC_BlockSize -1)   /* 1023 bytes */

#define RelocSlot(s)\
    index  = (int)(s) >> GC_BlockShift;                 \
    offset = (int)(s) &  GC_BlockMask;                  \
    base   = (int)(s) & ~GC_BlockMask;                  \
    for (tptr   = &RelocTable[index].Info;              \
         offset = tptr->Next_Block;                     \
         tptr   = (garbage-*)(base + tptr->Next_Block));\
         s = RelocTable[index].NewLoc + offset - tptr->Delta;
```

4.7 Post Processing

After the garbage-collection process is essentially over, PCs saved in activity stacks are restored to their original values. The EP (Environment Pointer - also a VAX register) saved with each PC on an activity stack frame is used to find the beginning of the relocated code vector object as follows:

The first reference in an environment vector object (what the EP points at) points at the corresponding method object. From the method object, a reference to the code object can be extracted. Using the process described above, this pointer will be redirected, added to the currently saved PC value (an offset into the code vector object), and then restored in the activity stack with the new value.

After this process of fixing up saved PCs. the only remaining task for the garbage collector is to clear the table so it can be used as a "to mark" stack the next time the mark phase is initiated as part of garbage-collection.

5 Compilation

The Trellis compiler provides an interactive mode, a debug mode as well as a batch mode for compiling Trellis *type modules*. Type modules define types, which contain operations, fixed names and components. The Trellis compiler is an *incremental* compiler, which implies a rapid turnaround on most compiles(since it retains a lot of information in a *program database*). The compiler allows the coexistence of erroneous code with correct code and incremental correction of code to create a functioning program. For a discussion of the major organizational units in the Trellis compiler and the compilation process as well as some details on how inheritance is implemented, please refer to [kilian87]. For a discussion of algorithms used in the compiler to optimize operation invocation, perform in-lining of operations and efficiently manage virtual memory, please refer to [kilian 88]. In this chapter we will try to take a more detailed look at how the compilation process described in [kilian87] is really achieved. We will describe the data structures used for managing source code, symbol tables used to manage type hierarchy, inheritance and secret inheritance rules, and precedence rules used for identifying specific operations(there may be multiple definitions of an operation in a type module, creating a name conflict that needs to be resolved).

5.1 The String Canonicalization Table

Compilation in Trellis involves managing the many different type names, operation names and fixed names that are present in the various type modules. To do efficient lookup of these symbols, symbols are hashed into a table(a *String Table*) at compile time and *canonicalized* to give a unique SID (*String IDentifier*). Hashing is used here to identify the specific bucket or slot in the string table that the string will be placed in, and canonicalization is the encoding of the string's location in the table (i.e. which bucket it falls into, and its position in the list or chain that starts off of that bucket).

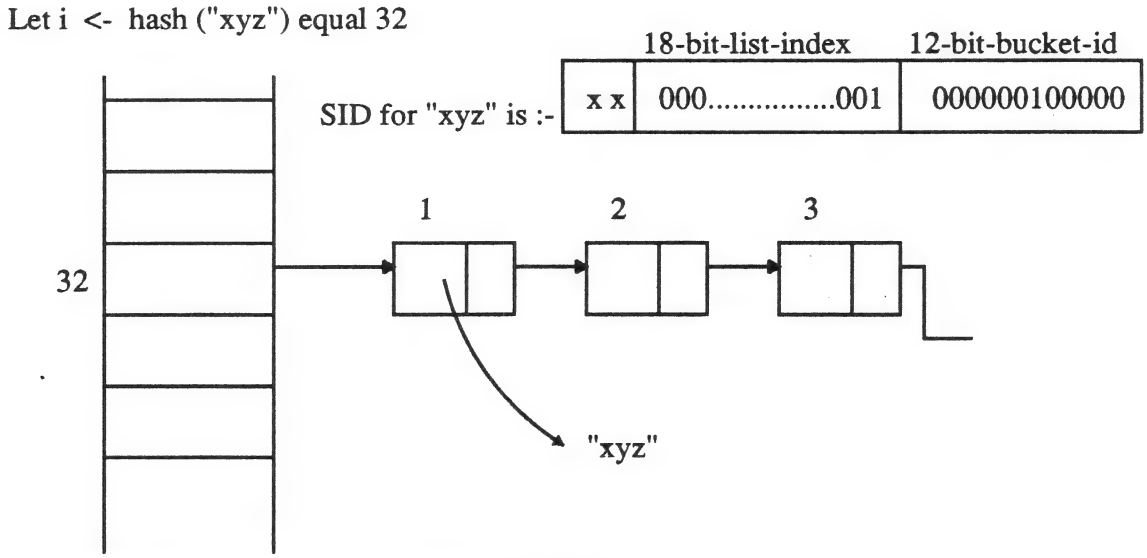


Figure 5-1 The String Canonicalization Table

There is one String Table per workspace. These SIDs are subsequently used wherever symbol

lookup becomes necessary at run-time. The compiler uses SIDs in generated code. Comparison of strings is reduced to matching their SIDs (integers). Each Type-module in Trellis has a symbol table of its own, where the canonicalized SIDs are used to place and retrieve the information for a symbol We now describe the canonicalization process:

The SID for a string is *unique*, has the form :- | XX | 18 bit hash chain index | 12 bit bucket id | (Here the two high bits, marked XX are not significant).The lower order 12 bits is the bucket into which a string gets hashed. The next 18 bits are used to identify the position(index) in a specific hash chain for the string. The string table size is 2**K where K <= 12. Thus canonicalizing a string such as "xyz" involves the following steps:

- i <--- hash("xyz") mod 2 ** 12 (this gives us the low order 12 bits)
- If the string is already present in the table, note the hash chain index and skip the next step, else create a new node for the string, which has a pointer to the string and a next pointer to get at the next node in the hash chain.
- Insert this node at the end of the chain, and note the position of the node in the chain. (the hash chain index).
- Perform the encoding of the position of the string in the string table into the format shown above. We now have the SID for the string.

The Type Symbol Table is structured with the nodes organized according to the SID of the primary symbol that the node has information on. The table size is 2**L where L <= 12 We find the bucket for an SID S by doing S mod 2**L and then compare SIDs until we reach the node that we want. A specific node may contain multiple definitions for the same SID. This happens when we have different versions of an operation (*public, private* etc). They are recorded in one node since they have the same symbolic name

We will now go through the compilation process for an example type module which has some operations and components. We will show the various stages of the compilation process, and try to provide more detail to some design issues discussed in [kilian87]. In Trellis, a definition is the smallest unit of source that can be compiled. Trellis operations, fixed names and components are definitions that can be compiled into object code and executed. We will discuss the various actions of the compiler in transforming a definition into compiled code.

5.2 The Compilation Process

Consider a type module which is as follows:-

```
type_module X

operation Z(me)
   returns (integer)
   signals (a, b, c)
is
begin
     ...
     ...
end;
end type_module;
```

Figure 5-2 Example Type Module

Figure 5-3 shows some system data structures that are updated after the definitions in the example type module have been parsed and the source code stored in source objects.The entire piece

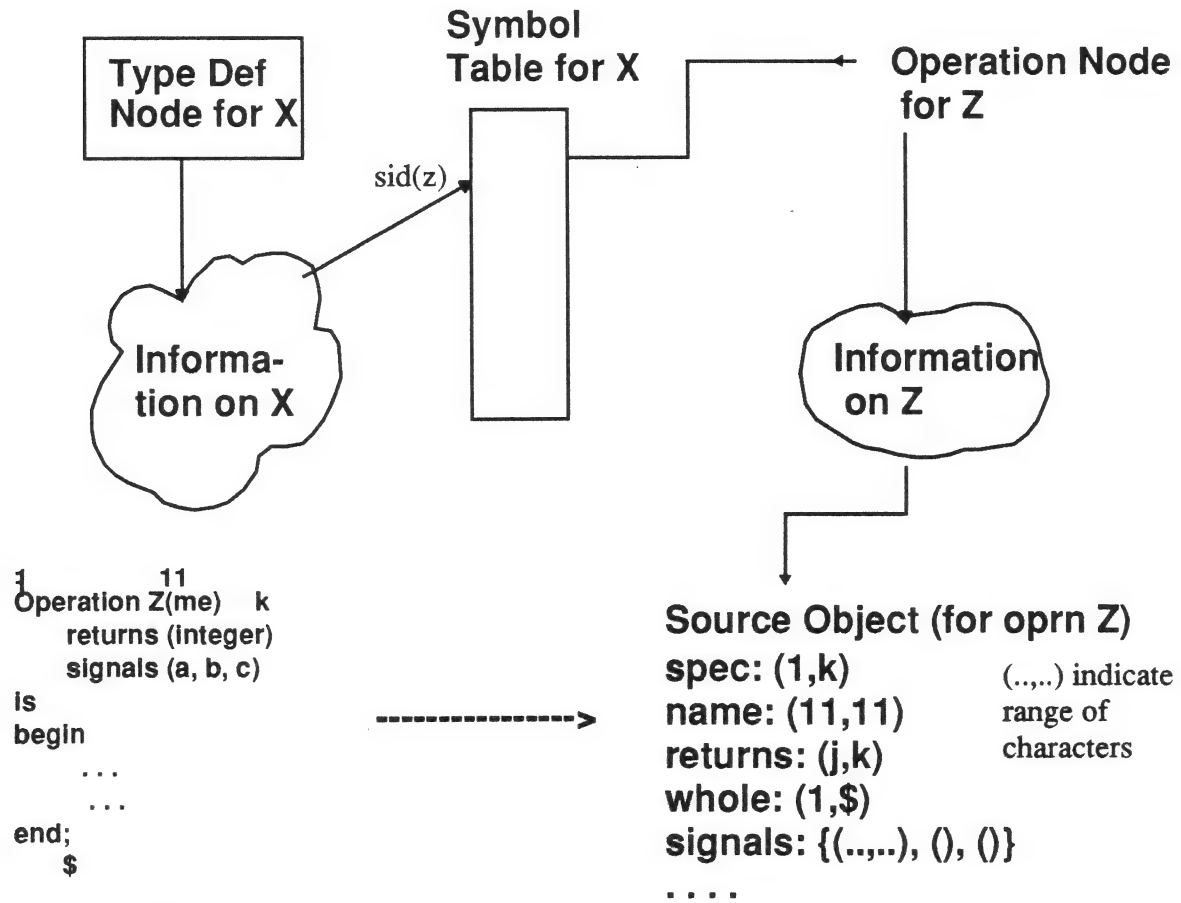


Figure 5-3 System Data Structures for a Definition

of source that describes operation Z above becomes a single owl string object with lines of text delimited by newline characters. Each definition has a set of elements that characterize it, its name, its source and its parse tree. As the compiler processes a definition, the various elements of the definition are placed in the appropriate system data structures (such as definition nodes for the type and the operation, a source code object etc). The last stage of compilation is when the compiler generates object code for a definition. The object code goes into associated method objects, code vector objects etc.

5.3 Compilation States and Actions

Definitions while being processed go through a series of states. When a definition has gone through all the states, it implies that it is consistent with the rest of the definitions and has had object code generated for it. The states that a definition goes through are *not_processed*, *parsed*, *inherited*, *analyzed* and *generated*. The compiler can process many definitions in parallel, and they may be in different states. A definition may be stalled at a particular state because of various kinds of errors. We now describe each of the states that a definition goes through, and the processing that is done in that state:

- *not_processed* : this is the initial state. The definition is yet to be processed.
- *parsed* : the source code is syntactically correct and all references to types and elements are resolved.
- *inherited*: the element has been inherited into the subtypes of the type module that this element belongs to.
- *analyzed*: the element has an implementation that is free of errors.
- *generated:* code was generated for the element.

Adding a Definition (states are shown in italics and actions in bold letters)

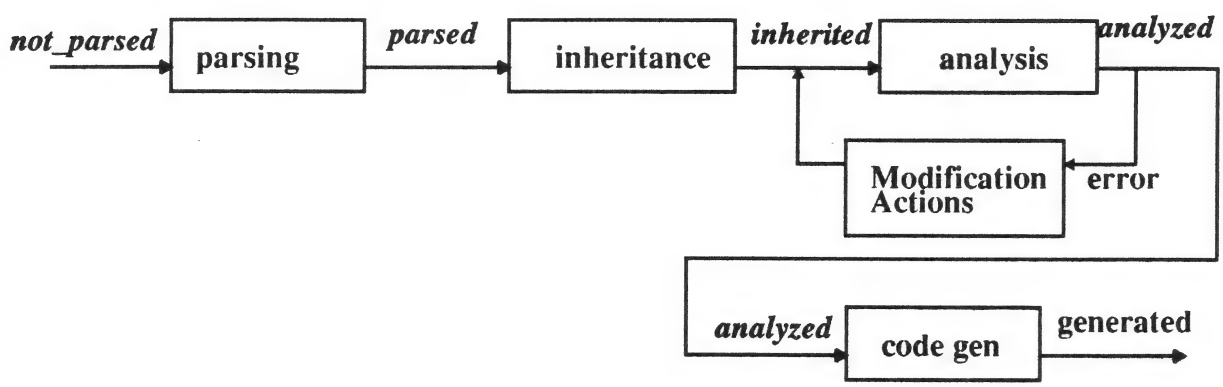


Figure 5-4 Definition States and Actions

The compiler can also be viewed as a set of actions that transforms definitions from one state to

the next state. Actions are independent of one another. They may also be combined in different ways depending on whether a definition is being added, modified or deleted. Actions are *idempotent*. This means that applying an action on a definition repeatedly would produce the same result. Actions also require the definitions that they can work on to be in their domain, i.e. the action will happen only if the definition is in one of a set of *proper* states.

Figure 5-4 shows the actions that are performed for a typical programming task: *adding* a definition. *Deleting* a definition involves two actions, removing all references to the element from the program database and reanalysis of definitions that reference the deleted element. The *modification* actions consist of removing the element from the program database (same as in deletion), adding the replacement element and reanalysis of all definitions that had referenced the old element (references are redirected to the new element). The various actions are managed by using the action recorder and the action sequencer. The action recorder maintains a list, the action list, of actions that have been performed and definitions that have been affected. The action sequencer uses the action list to determine what needs to be done to make a definition consistent. In the next section we look at the "type checking" action in detail, since this is one of the more involved actions in the compiler.

5.4 The Type Checking Action

This section describes the type checking action in the compiler. [Schaffert et al 85] is the language reference manual for Trellis. It describes how to define and implement new types as well as families of types in Trellis. It specifies the available attributes for types, and the rules for specifying operations and components in a type. There are constructs to specify different visibility extents for operations, components, and fixed names, subtype compatibility rules, and the semantics of inheritance for operations, components and fixed names present in multiple supertypes of a type. It also defines the scope of type names and fixed names. Trellis type definitions that deal with families of types include type generators (e.g. Set[T]) and union types. We will now look at the structures used by the compiler to manage all the type information. We will assume knowledge of the rules for type definitions as described in the language reference manual.

When a new type definition is to be compiled in to the workspace, the Trellis compiler creates a set of nodes called *definition nodes* for the type module itself, as well as components, operations, fixed_names that are present in the type module. We now look at the definition node for the type module, which has a number of fields and slots(the other definition nodes are similar, though they do have other fields that are specific to them). The fields are used as described below:

(a) An *attributes* byte is present which has bits in it that are set ***on*** or ***off***. There are bits to indicate if the type is ***constant*** (i.e. instances of the type have no time-varying properties) or ***mutable***, whether the type is a ***base*** type (i.e. there will be instances of this type) or an ***abstract*** type (this is when all objects of this type actually are instances of one of its subtypes). The ***nosubtypes*** bit is set when this type is specified as having no subtypes. There is a bit to indicate if this is a ***builtin*** type. The ***not_flat*** attribute bit is used while type-checking type-generators passed as parameters.

(b)a number of short integers are used to track the ***depth*** of this type (the depth is calculated by

going up the complete list of supertypes of this type), record the maximum number of parameters used in this type definition (***max_params***), keep a count of instance components, count of class components, and a value that is the unique SID for this type name.

(c) there is a list of immediate supertypes of this type module definition, list of ***all*** supertypes of this type module, a list of components, operations and fixed names that are ***secretly*** inherited into this type module from the supertypes (the **secret symbol table**), list of subtypes of this type module, as well as a ***bind list*** and a ***cross reference list***.

Bind lists are used to keep track of different resolutions for symbols that are shared between definitions (e.g. through inheritance). They help to determine the actual definition referred to by a symbol or operation invocation. Every definition node has a bind list. While processing any definition, the global variable Current_Bind_List should be set to the bind list for that definition. If the definition is a "copy"of a supertype's definition, Source_Bind_List (another global variable) should be set to the bind list of the definition from which this one originated; otherwise Source_Bind_List should be equal to Current_Bind_List.

The first element of bindlist contains a pointer to the definition (x) that contains it. Whenever something (y) is entered into this bindlist, it means that x references y and that the cross reference list of y should contain (x). Thus a cross reference list is basically a list of what all definitions use this specific definition. If this definition gets modified or changed, the definitions in the cross reference list need to be reanalyzed.

(d) a symbol table (the **type symbol table**) that contains all the symbols in the current type module that are significant (and need to be checked for name conflicts). The table will have parameters in the current scope, private fixed names, operations and components etc. The symbols are stored in a canonicalized form. [Needs further clarification here].

Consider the following example:

```
T :    foo        private
                  XXX(me)   public
                  foo(me)
                  YYY(me)   public

S :               YYY(me)    {override}
                   T'YYY(me)
     (inherited):XXX(me)   <=> T'XXX(me)
    (secret inh):foo(me)   where me.basetype = S
```

The symbol table in S is organized so that the string "YYY" in S is first the public instance YYY and then the qualified instance T'YYY. The operation foo(me) in S cannot be directly invoked. It is placed in the secret table (which is really a list), and is not present in the symbol table.

(e) there is a reference to the source object for this definition as well a reference to the environment data (which is placed in a separate object accessible through this definition. Additionally there is a pointer to an object that is the name of the type module. (?)

6 Execution

A majority of Trellis code invloves invoking an operation. Unfortunately, it is not always determinable at compilation time which operation is being invoked because it depends on the type of the controlling argument (which may not be known until execution time). Thus, sometimes an *operation apply* must be used at execution time to determine which operation is being invoked. Earlier (in Chapter 5, *Compilation)* it was explained how the compiler can avoid operation applies and use a *direct call* instead. In this chapter, we will explain exactly how a direct call works (not quite as simple as it sounds) and also how an operation apply is implemented when it can not be avoided.

6.1 Operation Implementation

In Chapter 2, *Storage Representation of Objects*, three kinds of objects were introduced that are generated by the compiler to implement Trellis operations: method objects, code vector objects, and environment vector objects. This was designed to separate the object references from the executable code for purposes of garbage collection. In other words, the garbage collector needs to know how to locate all object references in order to mark all accessible objects and to modify those object references if a referenced object is relocated. Allowing object references to exist at arbitrary places in objects with executable code was not a viable design alternative; therefore, these three kinds of objects were chosen to implement operations and avoid this problem.

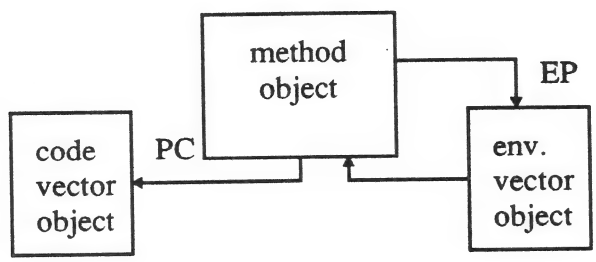


Figure 6-1 Three Kinds of Objects to Implement Operations

In short, the code vector object contains only executable code (in the *bytes* section) and the environment vector object contains only references to other objects (in the *refs* section) which are accessed by the corresponding code vector object. The method object puts the absolute address (an object reference) of the corresponding environment vector object into a register (the *EP* or Environent Pointer) and then transfers control to the code vector object's executable code. Both of these actions are accomplished by imbedding the absolute addresses (object references) of the corresponding code and environment vector objects in the method object's executable code. This is described in more detail below.

This allows the garbage collector to treat environment vector and code vector objects uniformly; but, method objects are still special because there have references to other objects (the code and environment vector objects) in the bytes section. However, the location of these references is known.

As background for our discussion of an operation invocation, we now describe how registers are used, conventions for passing arguments to an retrieving return values from an operation, and the actual instructions generated by the Trellis compiler for a direct call.

6.2 Register Usage

- o R10 is the LP (Locals Pointer) - this points at the first of an operation's local variables on the current stack.
- o R11 is the EP (Environment Pointer) - this points to the environment vector object associated with the current operation.
- o AP is the Argument Pointer - this isn't used. Normally, this would point to the first argument on the stack. This is obviated by the use of the LP. R4-R8 are also not used.
- o FP is the Frame Pointer - this is used to point to a valid frame on the system stack.
- o SP is the Stack Pointer - it points to the last byte allocated on the current stack.
- o PC is the Program Counter - it points to the first byte of the next instruction to execute.

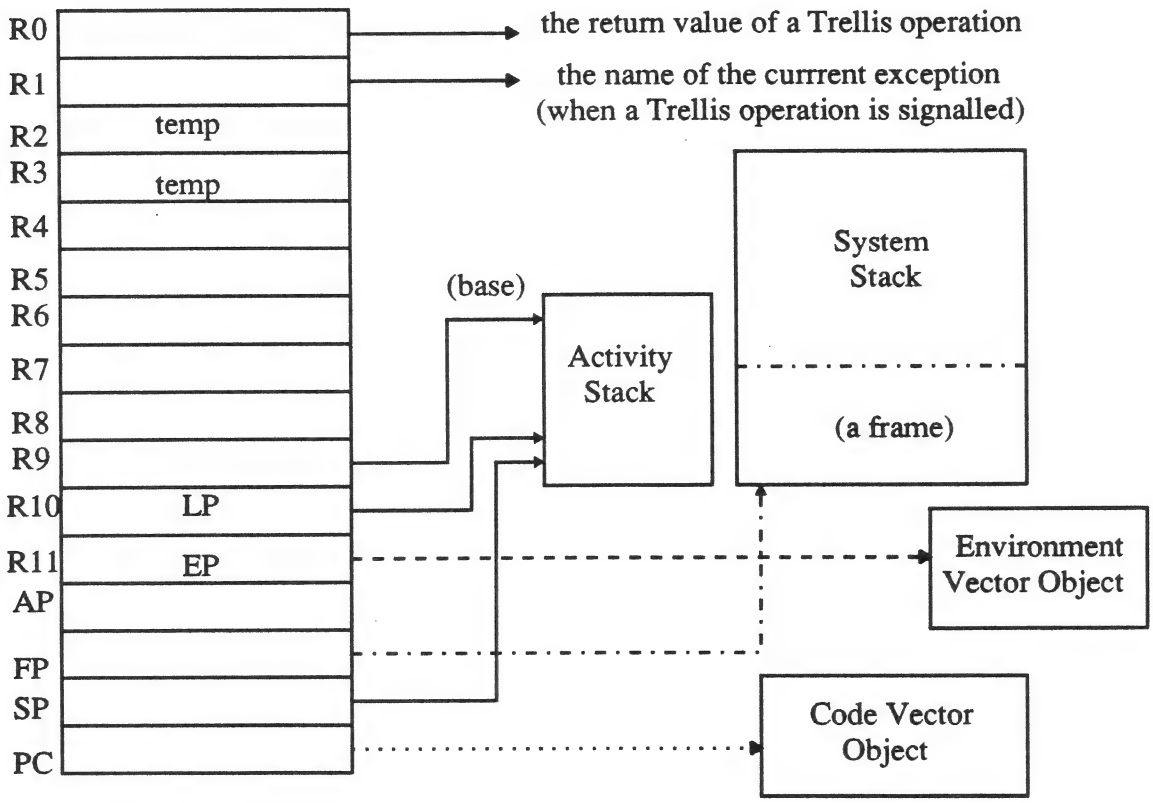


Figure 6-2 Register Usage

6.3 Argument Passing

The VAX calling standard states that any arguments to a procedure will be pushed onto the stack in the opposite order than they were specified. The Trellis calling standard also requires that arguments to an operation will be passed on the stack; but, they are pushed in the order that they are specified (*not* in reverse order). Thus, a VAX procedure `P` with arguments `a` and `b` [`P(a,b)`] will push `b` onto the stack and then push `a` whereas a Trellis procedure `Q` with arguments `c` and `d` [`Q(c,d)`] will push `c` and then `d`.

An operation's arguments (references to objects) are pushed onto the stack as longwords before an operation invocation (with the PUSHL instruction). After an operation invocation, the stack will be popped by incrementing SP by the number of actual arguments pushed times the size of each (a longword or 4 bytes). If n arguments had been pushed, this would be accomplished with an instruction like the following:

```
ADDL2  4*n,SP    ! Pop n arguments off of the stack
```

Figure 6-3 Code to Pop n Arguments off of the Stack

6.4 Return Values

As we described above, register R0 is used to hold the return value for an operation (an object reference). So, for most operations (those that return a value), the last part of an invocation of that operation would be to take the called operation's return value (stored in R0) and place it in a destination (usually a local variable).

Consider the following line of Trellis code that invokes operation F with two arguments x and y and returns the result of F in z:

```
z := F(x,y);
```

Figure 6-4 An Operation Invocation in Trellis

Here is some of the code that would be generated (and placed in a code vector object) for the operation invocation shown above:

```
PUSHL  x(LP)       ! Push x onto the stack
PUSHL  y(LP)       ! Push y onto the stack
!(see 6-6 below)   ! Call F with the arguments on the stack
MOVL   R0,z(LP)    ! Store F's return value in z
```

Figure 6-5 Partial Listing of Generated Code for an Operation Invocation

LP points to the first local variable on the stack and z is the offset from LP where the reference to object z is stored. So, in the last instruction of the sequence above, z(LP) is a specification for a reference to object z (a local variable). Similar specifications are used elsewhere in this document to indicate object references. Thus, this instruction will place the return value from F() (an

object reference stored in R0) into the variable z (the slot on the stack where the reference to object z is kept).

6.5 Preserving State

Before we can invoke another operation, we must understand what state needs to be preserved (volatile registers and the stack frame) to resume the current operation. Basically, there are three things to save: a pointer to the next instruction to execute (the PC), the arguments to the operation (accessed off of the LP), the local "variables" and temporary objects (also accessed off of the LP), and a pointer to a set of references to "global" objects (like other operations) that this code will reference (these are all stored in the environment vector object which the EP points to).

These three registers (the LP, EP, and PC) will be saved, reused, and restored across operation invocations. Because the SP is used to implement operation invocations (it points to the top of the stack where these other registers will be saved and restored), it is not saved and restored (it is always in use). Other registers need not (and will not) be saved because they are only used as temporaries for expression evaluation (but not across operation invocations). The compiler also uses slots on the stack as references to temporary objects (which might be used to hold intermediate results between operation invocations).

Realize that the contents of an object may change across operation invocations.

6.6 Operation Invocation

We now describe the actual operation invocation (a direct call) which occurs after the arguments to the operation have been pushed onto the stack. In order to invoke the operation F, we must load the address of F's method object from the environment vector object into a register. Finally, we need to transfer control to F's method object. This is accomplished with the following instructions:

```
MOVL k(EP),R2 ! F's address is k bytes into the env. vector
JSB  HL(R2)   ! Call F and skip its object header (HL = 8)
```

Figure 6-6 Code Generated for a Direct Call

The stack now has the actual parameters for operation F(). The call will also push the PC (the return address - the next instruction to execute upon returning from F) onto the stack. We now present the standard prolog (a common set of instructions) for a method object:

```
MOVQ LP,-(SP) ! Pushes LP and EP onto the stack
MOVL SP,LP    ! Reset  LP to the beginning of F's locals
MOVL I^#EV,EP ! Set EP to the environment vector
JMP  @#CV+HL  ! Jump to F's code vector (skip the header)
```

Figure 6-7 Standard Prolog for a Method Object

The only thing that is different for each method object is the address of the corresponding environment vector and code vector objects (EV and CV).

In this prolog code, notice that the PC has already been pushed. First, the LP and EP are pushed (after the PC). The LP was reset to point to the top of the stack (the beginning of F's local variables). EP was set to point to the corresponding environment vector object. The JMP (Jump) instruction sets the PC to the first instruction in the code vector's bytes section.

Once inside the code object, the following kind of code is used:

```
SUBL2 k,SP      ! Make room on the stack for local variables
```

Figure 6-8 Code to Allocate Local Variables

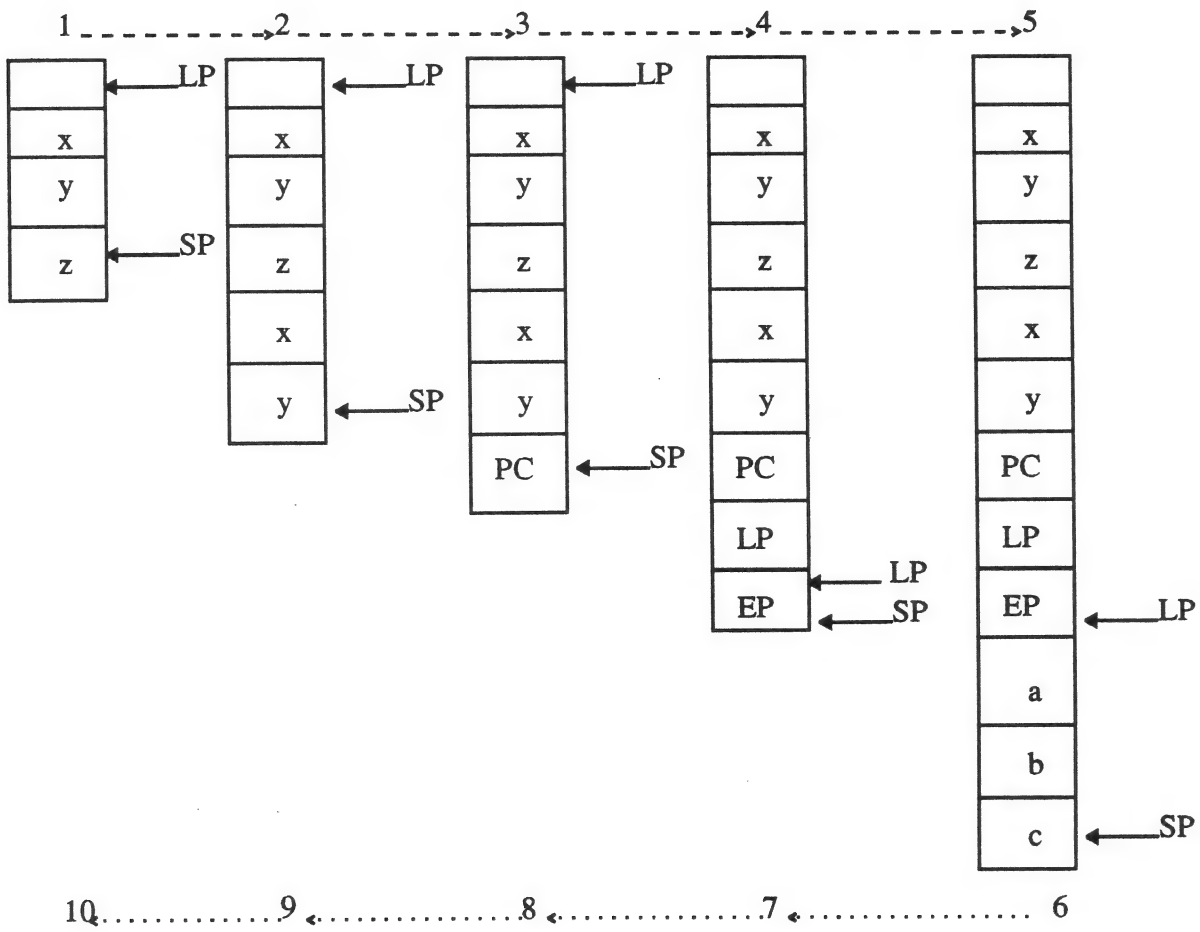


Figure 6-9 Stack States

The various stack states while invoking operation F are described below (1-5) and depicted in figure 6-9. The steps necessary to resume the operation that invoked F (after operation F has completed its execution) are also described below (6-10) and depicted in figure 6-9:

A. is the initial state of the stack with three local variables (x, y, and z).

B. is after the arguments to F have been pushed (x and y).

C. is after calling F (with the JSB instruction) - notice the PC is pushed.

D. is after saving the LP and EP on the stack and resetting the LP (to point at the top of the stack - the same value as SP).

E. is after making room on the stack for three local variables used in F (a, b, and c).

F. The stack will be rolled back to pop all the local variables (set SP to LP).

G. The LP and EP will be popped and restored.

H. An RSB instruction will be executed (popping and restoring the PC).

I. The arguments to F will be popped.

J. The stack will now look like it did before operation F was invoked.

The code to return from an operation looks like the following:

```
MOVL <return-value>,R0
MOVL LP,SP       ! Roll back the SP to its previous level
MOVQ (SP)+,LP    ! Pop the LP and EP
RSB              ! Return to the caller (PC is popped)
```

Figure 6-10 Code to Return From an Operation

6.7 Operation Apply

Although 80% of operation invocations can use a direct call, 20% must still use an operation apply to determine the proper operation to invoke based on the type of the controlling argument. Trellis uses a few techniques to make operation applies execute as quickly as possible.

A. Inheritance is implemented via *copy-down* which localizes the searching (all possible operations that are candidates for invocation are in the symbol table of the controlling argument's type).

B. The operation names are hashed at compile time and are organized so those operations which might require an apply are listed first (this might happen with two operations with the same name and different controlling arguments - such as create(me) and create(mytype)).

C. There is an operation cache created for every invocation site that doesn't use a direct apply. Whenever an operation apply is done, the type of the controlling argument is compared with the type stored in the cache. If they are the same, the corresponding operation (also stored in the cache) is invoked. If they are not equal, a class lookup operation is performed (to find the

proper operation to invoke). The results are stored in the cache for the next operation apply. Since one call site usually invokes the same operation, the cache hit rates have been very high.

7 Activities and Interrupts

This chapter on activities and interrupts will be written as soon as we have information on some recent design changes to the activity system.

8 Builtins

There are a number of categories of builtins in Trellis:

A. Builtin Operations - these are primitive Trellis operations not implemented in Trellis. Many operations in the Trellis run-time library are specified as interfaces to these builtin operations. Builtin operations are stored in a set of reusable code vector objects based at a fixed location. They are two kinds of these operations:

   1. external routines written in C which are called from the code-vector (a majority of builtin operations are implemented this way).

   2. routines written in VAX assembly language and implemented directly in the code vector (integer arithmetic operations are implemented this way). These routines are sometimes called "in-lines" because they don't contain a call to an external routine.

B. Builtin Fixed Names - these are predefined names (like True and False) with a corresponding value (a Trellis object).

C. "Intrinsic Functions" - these are special routines known to the compiler. For example, to implement an allocate operation (like `create(MYTYPE)`), the compiler will generate a call to the routine `Allocate()`.

D. World Variables - these are variables whose values persist in the workspace and which reference important builtin Trellis objects. These objects include: all builtin classes, some "special" builtin identifiers, and the builtin objects which are the values of the builtin fixed names (like True and False).

8.1 Builtin Operations

For purposes of discussion, let us examine an ordinary (non-builtin) Trellis operation definition:

```
operation foo( me, arg1: Type_of_arg1, .... )
     is
     begin
          ! Trellis code goes here
          -
          -
     end;
```

Figure 8-1 An Ordinary Operation Definition

As has been described elsewhere, when this operation definition is successfully compiled, the compiler will have created a method object, an environment vector object, and a code vector object.

If `foo` was not a user-defined operation, but a builtin operation instead, it would be specified as follows:

```
operation foo( me, arg1: Type_of_arg1, .... )
     is
     builtin(32);
```

Figure 8-2 Specification of a Builtin Operation

The only difference in the specification of the user-defined operation `foo` and the builtin operation `foo` is that the builtin operation has no body but only "`builtin(32)`".

When the builtin operation `foo` is successfully compiled, the compiler will have generated an empty environment vector object and a method object with a pointer to the code vector object for builtin operation #32. Realize that no *new* code vector object is generated; a method object will be created which will point to an *already existing* code vector object.

Notice that builtin operations are specified with a numeric identifier rather than a symbolic name. This facilitates a table lookup by the compiler to find the address of a builtin operation's code vector object. The Trellis compiler will not compile a reference to a non-existant builtin operation. The numeric identifiers of builtin operations start at 0 and range consecutively to a maximum number (the number of implemented builtin operations that were linked with the Trellis system).

The code for a builtin operation can be written in any VAX language and is compiled and linked with the rest of the Trellis system. So, the implementation of a builtin operation must be provided when the Trellis system is linked; but, the Trellis interface to that builtin operation is compiled by the Trellis compiler and stored as part of the workspace. Most library operations are implemented in this manner.

8.2 Builtin Fixed Names

Fixed names are symbolic names for values determined at compilation time. These are similar to constants in other languages. Most predefined types (also called builtin classes) have basic values which are referenced via these builtin fixed names. For example, the builtin class Boolean has two predefined fixed names: True and False. These fixed names refer to the predefined objects True and False which are used to implement boolean operations.

8.3 Intrinsic Functions

As opposed to builtin operations, these functions are known by the compiler. The most common example of an intrinsic function is the routine `Allocate()` which is used to implement allocate operations (like `create(MYTYPE)`).

The Trellis system has variables which contain the addresses of these functions and are used by the code generator and the code lister. To accomplish this, the address of each intrinsic function

is fixed when the Trellis system is linked. Because they are referenced with a symbolic name, the compiler needn't perform a table lookup to find their address.

These functions are also sometimes called "in-lines". They are not like other Trellis operations because they don't have an interface specified in Trellis, do not follow the same calling conventions, and are not implemented with the method, environment vector, and code vector objects.

8.4 World Variables

All other kinds of builtins (classes, special identifiers, and objects) are referred to by these variables. These variables are stored in the world header which is part of the workspace (so they persist). These variables exist so that the Trellis system can refer to these objects with the same ease that they are referred to in Trellis code.

Builtin classes are used to quickly compare an object's type with a known object type (e.g. method class). Builtin objects are used to compare an object reference with a predefined object (e.g. the True object).

9 Callout

Because Trellis does not conform to the VAX calling standard (for reasons of performance), a user can not write Trellis code to directly call external routines (those written in languages other than Trellis). Also, Trellis operations can not be directly called from external routines. To overcome these problems, the Callout facility was developed. There are two modules that constitute this facility: `callout_buffer` and `callout_facility`. Each module is introduced with a brief description below. A more complete description of the contents, usage, and implementation of these modules is provided in the following sections.

The `callout_buffer` module defines a type of buffer, similar to an untyped byte vector, to use in converting between the Trellis data formats and the VAX data formats. It is also used as a "container" to pass any type of Trellis data to an external routine. In order to pass a Trellis argument to an external routine, the appropriate `encode()` and `decode()` operations must be defined. These operations perform the necessary conversions and place the result into a callout buffer (which can then be passed as an argument to an external routine).

The `callout_facility` module provides the operations perform the actual call-out and call-in actions using callout buffers. These operations dynamically link the shared image containing the external routine with the Trellis system image, call that external routine, and return the external routine's return value in a variety of forms. Each operation in this module takes a fixed sequence of integers or callout buffers which represent the arguments to the external routine. They will be passed to the external routine (pushed onto the stack) in the opposite order that they are specified (which conforms with the VAX calling standard described in the chapter entitled *Execution).*

9.1 Callout Buffer

A callout buffer has three parts: a byte vector (in C this is a pointer to an array of bytes), a zero based offset (where to read or write the next byte), and a size (the number of bytes needed to represent the Trellis data type being passed). They may be used to contain the actual arguments to an external routine and/or to hold the routine's return value.

The offset in a callout buffer is used to concatenate multiple pieces of Trellis data into one piece of VAX data. An example of this is a complex number (a + bi).

Please note that although a callout buffer defined in C (as part of the Trellis system code) has these three parts and is equivalent to the Trellis object definition, a casual reference to a callout buffer (in this document and others) usually only one refers to one field or component (the array of bytes or the address of such).

There are various parameter passing mechanisms on VMS: by value, by reference, and by descriptor (usually for strings). Callout buffers are passed by reference and integers are passed by value (unless they are passed in a callout buffer). Boolean's can also be passed by value by converting them to an integer. A string descriptor can be created for a Trellis string and then put into a callout buffer (which will be passed by reference).

To place an actual argument to an external routine into a callout buffer, use the operation `by_ref()`. This operation should be defined for each type that could be passed as an argument to an external routine - it is not part of the `callout_buffer` module. It should create a callout buffer, encode the data in that callout buffer (using the appropriate `encode()` operation), and then return the callout buffer. Trellis integers can be passed as is and are automatically converted to VAX integers.

A callout buffer can also be used to hold the result of the external routine. Use the `create()` operation (defined in `callout_buffer`) to create a callout buffer for this purpose. Values can be extracted from a callout buffer via the appropriate `decode()` operation.

9.2 Callout Facility

There are a number of operations that are defined in this module. They fall into four categories (all categories except the third only have one operation):

A. `callout()` - returns a callout buffer (which must be decoded) which contains the values held in registers R0 and R1 upon the completion of the external routine.

B. `callout_status()` - returns the completion status of the external routine (some typed code indicating success or failure) as a Trellis integer (no decoding necessary). If the low bit of the completion status is clear (indicating failure on VMS), the message corresponding to that status code is obtained (via `SYS$GETMSG`) and the appropriate signal is raised.

C. `callout_*()` - returns a particular type of Trellis object directly (these operations perform the appropriate decode actions automatically). These operations include `callout_integer()`, which returns a Trellis integer; and, `callout_f_real()`, `callout_g_real()`, and `callout_d_real()`, which return a Trellis real from a VAX F, D, and G real respectively.

D. `callback_routine()` - creates a callout_buffer which can be called from an external routine to invoke a particular Trellis operation. The details of this operation are explained in the next section.

All of these operations share a section of code (`Callout_Call`) which is implemented on VMS via the `RTL` routine `LIB$FIND_IMAGE_SYMBOL`. The arguments to this routine are a name of a shareable image and the name of a universal symbol (which is the external routine to call-out to).

To create a shareable image which contains the external routine to call, use the `/SHARE` qualifier on the `LINK` command with an options file that declares the external routine name as a universal symbol.

To perform a call-out, the shareable image is dynamically "activated" (linked and loaded) in the P0 segment of the process's address space. The address where the universal symbol has been loaded is returned and is used to call that external routine.

9.3 Call-In

There has been some experimental code developed as part of the Trellis system in order to facilitate "Call-In". This phrase is used to describe the action of an external routine (written in any VAX language) invoking a Trellis operation. The steps necessary to use this facility are described below:

A. Write a Trellis operation that you want to invoke from an external routine.

B. Write the external routine that will invoke that Trellis operation and create a shareable image with this routine (see the paragraph above). The semantics of the invocation are given the address of a routine, call that routine.

C. Create the arguments for `callback_routine`():

   1. Put the Trellis operation (to invoke) in the form of a `declared_operation` by calling `lookup_public()`.

   2. Create a vector of objects which contain the arguments to the Trellis operation.

   3. Create a vector of objects which will contain the result of the Trellis operation.

D. Invoke `callback_routine` (with the arguments prepared above) which will return a callout buffer with a code vector which can be called via CALLS (a VAX procedure-call instruction). This code vector contains an entry mask and code which puts the address of the method to invoke in R1, the address of the argument vector in R2, and the address of the result vector in R3 and then jumps to `the` Trellis system routine `Callback()` (which eventually invokes `the` Trellis operation `Unchecked_Method_Call`()).

E. Obtain the address of this callout buffer via the `address()` operation.

F. Pass the address of the callout buffer to the external routine via one of the operations in the `callout_facility` module.

G. The external routine will now execute the code which will invoke the Trellis operation.

As indicated above, this is experimental code that may not work, is subject to change, and won't work if the area containing the operation's method object is garbage collected and the method object is relocated.

```
void callin(op)
void*op();
   {
   printf("This is callin.\n");
   (*op)();
   }
```

Figure 9-1 C code to invoke a Trellis operation (via call-in)

10 Documentation

Getting Started with Trellis --- to get one started on using the Trellis/Owl programming language and the Trellis programming environment.

Trellis Installation Guide --- explains how to set up your VAX so that you can run Trellis, and how to install the files that make up the Trellis system.

Release Notes --- lists changes since the last release, and notes omissions from the documentation.

Using Trellis/Owl from DCL --- describes how to invoke the Trellis/Owl language system from DCL, and how to compile and run programs without using the Trellis environment.

The Trellis User Interface --- describes how the keyboard and mouse are used with Trellis. Explains making selections, pushing buttons, text editing, and so forth.

The Trellis Programming Environment: User Guide --- gives detailed descriptions of each tool provided by Trellis.

Overview of the Trellis/Owl Predefined Library --- describes the library of basic types supplied in the LIBRARY.WSP workspace. Includes strings, vectors, input/output, and so forth. Does not document the graphics and user interface types which are added in the {\tt TRELLIS.WSP} workspace.

Introduction to Activities and Concurrency Control in Trellis --- describes how to use the Trellis multiple-activity system.

Trellis Callout Facility Guide --- describes how to call out from Trellis/Owl to other programming languages and libraries.

The following technical reports are also available:

An Introduction to Trellis/Owl --- a brief overview of the Trellis/Owl language. Recommended for familiarizing yourself with the language. DEC-TR-438.

The Trellis Programming Environment --- an overview of Trellis. Helpful for familiarizing yourself with the capabilities of the environment. Does not reflect the latest versions of many of the Trellis tools. DEC-TR-503.

Trellis Object-Based Environment: Language Reference Manual --- a complete reference manual for Trellis/Owl. DEC-TR-372.

Trellis Object-Based Environment: Language Tutorial --- a tutorial introduction to Trellis/Owl. Note that some examples use hypothetical operations and types that are not necessarily present in the supplied libraries. DEC-TR-373.

Using Types and Inheritance in Object-Oriented Programming --- a general introduction to some

principles of object-oriented programming. Reprinted from *IEEE Software*. (A revised version of DEC-TR-437, *Using Types and Inheritance in Object-Oriented Languages.*)

Concurrency Features for the Trellis/Owl Language --- describes the Trellis activity system, its use, and its implementation. DEC-TR-439.

Persistent and Shared Objects in Trellis/Owl --- describes ongoing research into extending Trellis/Owl to include object-oriented database capabilities. DEC-TR-440.

11 References

[O'Brien 85] Patrick O'Brien. *Trellis Object-Based Environment: Language Tutorial*, Digital Equipment Corporation Technical Report DEC-TR-373, November 1985.

[Schaffert *et al* 85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment: Language Manual.* Digital Equipment Corporation technical report DEC-TR-372, November 1985.

[Halbert and O'Brien 86] Daniel C. Halbert and Patrick O'Brien. Using types and inheritance in object-oriented languages. Digital Equipment Corporation technical report DEC-TR-437, April 1986.

[O'Brien *et al* 86] Patrick O'Brien, Bruce Bullis, and Craig Schaffert. Persistent and Shared Objects in Trellis/Owl. *International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986. DEC-TR-440.

[Schaffert *et al* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. *OOPSLA86 Proceedings*, Portland, Oregon, September 1986. DEC-TR-438

[O'Brien et al 87] Patrick O'Brien, Daniel C. Halbert and Michael F. Kilian. The Trellis Programming Environment. OOPSLA87 Proceedings, Orlando, Florida, October 1987.DEC-TR-503

[Halbert 87] Daniel C. Halbert. The User Interface of the Trellis Programming Environment. March 1987.

[Moss and Kohler 87] J. Eliot B. Moss and Walter H. Kohler. Concurrency features of the Trellis/Owl Language. *ECOOP87 Proceedings*, Paris, France, June 1987. DEC-TR-439

[Kilian 87] Michael F. Kilian. An Overview of the Trellis/Owl Compiler. DEC-TR-504, November 1987

[Kilian 88] Michael F. Kilian. Why Trellis/Owl Runs Fast. March 1988